Foundations of Natural Language Processing
Lecture 4
Language Models: Evaluation and Smoothing

Alex Lascarides
(Slides based on those from Alex Lascarides, Sharon Goldwater and Philipp Koehn)

24 January 2020

School of informatics

Recap: Language models

- **Language models** tell us $P(\vec{w}) = P(w_1 \dots w_n)$: *How likely to occur is this sequence of words?*

  Roughly: *Is this sequence of words a "good" one in my language?*

- LMs are used as a component in applications such as speech recognition, machine translation, and predictive text completion.

- To reduce sparse data, N-gram LMs assume words depend only on a fixed-length history, even though we know this isn't true.

Evaluating a language model

- Intuitively, a trigram model captures more context than a bigram model, so should be a "better" model.

- That is, it should more accurately predict the probabilities of sentences.

- But how can we measure this?

Two types of evaluation in NLP

- **Extrinsic**: measure performance on a downstream application.
  - For LM, plug it into a machine translation/ASR/etc system.
  - The most reliable evaluation, but can be time-consuming.
  - And of course, we still need an evaluation measure for the downstream system!

- **Intrinsic**: design a measure that is inherent to the current task.
  - Can be much quicker/easier during development cycle.
  - But not always easy to figure out what the right measure is: ideally, one that correlates well with extrinsic measures.

Let's consider how to define an intrinsic measure for LMs.

Entropy

- Definition of the **entropy** of a random variable X:

  $H(X) = \sum_x -P(x)\ \log_2 P(x)$

- Intuitively: a measure of uncertainty/disorder

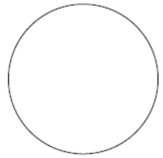
- Also: the expected value of $-\log_2 P(X)$

Entropy Example

One event (outcome)

$P(a) = 1$

$$\begin{aligned} H(X) &= -1 \log_2 1 \\ &= 0 \end{aligned}$$

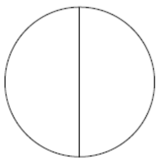


Entropy Example

2 equally likely events:

$P(a) = 0.5$
$P(b) = 0.5$

$$\begin{aligned} H(X) &= -0.5 \log_2 0.5 - 0.5 \log_2 0.5 \\ &= -\log_2 0.5 \\ &= 1 \end{aligned}$$

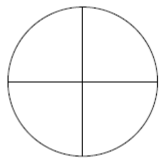


Entropy Example

4 equally likely events:

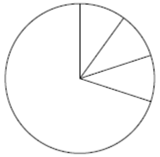
$P(a) = 0.25$
$P(b) = 0.25$
$P(c) = 0.25$
$P(d) = 0.25$

$$\begin{aligned} H(X) &= -0.25 \log_2 0.25 - 0.25 \log_2 0.25 \\ &\quad - 0.25 \log_2 0.25 - 0.25 \log_2 0.25 \\ &= -\log_2 0.25 \\ &= 2 \end{aligned}$$



Entropy Example

$P(a) = 0.7$
$P(b) = 0.1$
$P(c) = 0.1$
$P(d) = 0.1$

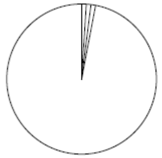


3 equally likely events and one more likely than the others:

$$
\begin{aligned}
H(X) = & -0.7\log_2 0.7 - 0.1\log_2 0.1 \\
& -0.1\log_2 0.1 - 0.1\log_2 0.1 \\
= & -0.7\log_2 0.7 - 0.3\log_2 0.1 \\
= & -(0.7)(-0.5146) - (0.3)(-3.3219) \\
= & \, 0.36020 + 0.99658 \\
= & \, 1.35678
\end{aligned}
$$

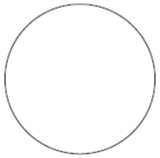
Entropy Example

$P(a) = 0.97$
$P(b) = 0.01$
$P(c) = 0.01$
$P(d) = 0.01$

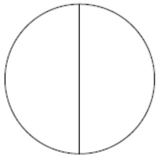


3 equally likely events and one much more likely than the others:

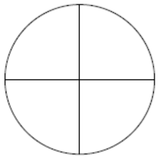
$$
\begin{aligned}
H(X) = & -0.97\log_2 0.97 - 0.01\log_2 0.01 \\
& -0.01\log_2 0.01 - 0.01\log_2 0.01 \\
= & -0.97\log_2 0.97 - 0.03\log_2 0.01 \\
= & -(0.97)(-0.04394) - (0.03)(-6.6439) \\
= & \, 0.04262 + 0.19932 \\
= & \, 0.24194
\end{aligned}
$$



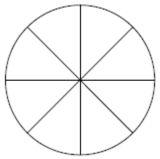
$H(X) = 0$



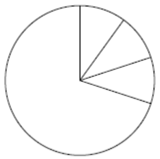
$H(X) = 1$



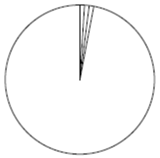
$H(X) = 2$



$H(X) = 3$



$H(X) = 1.35678$



$H(X) = 0.24194$

Entropy as y/n questions

How many yes-no questions (bits) do we need to find out the outcome?

- Uniform distribution with 2^n outcomes: n yes-no questions.

Entropy as encoding sequences

- Assume that we want to encode a sequence of events X.
- Each event is encoded by a sequence of bits, we want to use as few bits as possible.
- For example
  - Coin flip: heads = 0, tails = 1
  - 4 equally likely events: a = 00, b = 01, c = 10, d = 11
  - 3 events, one more likely than others: a = 0, b = 10, c = 11
  - Morse code: *e* has shorter code than *q*
- Average number of bits needed to encode $X \geq$ entropy of X

The Entropy of English

- Given the start of a text, can we guess the next word?
- For humans, the measured entropy is only about 1.3.
  - Meaning: on average, given the preceding context, a human would need only 1.3 y/n questions to determine the next word.
  - This is an upper bound on the true entropy, which we can never know (because we don't know the true probability distribution).
- But what about N-gram models?

Coping with not knowing true probs: Cross-entropy

- Our LM *estimates* the probability of word sequences.
- A good model assigns high probability to sequences that actually have high probability (and low probability to others).
- Put another way, our model should have low uncertainty (entropy) about which word comes next.
- **Cross entropy** measures how close $\hat{P}$ is to true P:

  $H(P, \hat{P}) = \sum_x -P(x)\ \log_2 \hat{P}(x)$
- Note that cross-entropy $\geq$ entropy: our model's uncertainty can be no less than the true uncertainty.
- But still dont know $P(x)$...

Coping with Estimates: Compute per word cross-entropy

- For $w_1 \ldots w_n$ with large n, per-word cross-entropy is well approximated by:

$$H_M(w_1 \ldots w_n) = -\frac{1}{n} \log_2 P_M(w_1 \ldots w_n)$$

- This is just the average negative log prob our model assigns to each word in the sequence. (i.e., normalized for sequence length).
- Lower cross-entropy $\Rightarrow$ model is better at predicting next word.

Cross-entropy example

Using a bigram model from Moby Dick, compute per-word cross-entropy of $\mathrm{I\ spent\ three\ years\ before\ the\ mast}$ (here, without using end-of sentence padding):

$$
\begin{aligned}
& -\tfrac{1}{7}(\quad \lg_2(P(\textit{I})) + \lg_2(P(\textit{spent}|\textit{I})) + lg_2(P(\textit{three}|\textit{spent})) + \lg_2(P(\textit{years}|\textit{three})) \\
& \qquad + \lg_2(P(\textit{before}|\textit{years})) + \lg_2(P(\textit{the}|\textit{before})) + \lg_2(P(\textit{mast}|\textit{the})) \quad) \\
= \; & -\tfrac{1}{7}(\quad -6.9381 - 11.0546 - 3.1699 - 4.2362 - 5.0 - 2.4426 - 8.4246 \quad) \\
= \; & -\tfrac{1}{7}(\quad 41.2660 \quad) \\
\approx \; & \quad 6
\end{aligned}
$$

- Per-word cross-entropy of the *unigram* model is about 11.
- So, unigram model has about 5 bits more uncertainty per word then bigram model. But, what does that mean?

Data compression

- If we designed an optimal code based on our bigram model, we could encode the entire sentence in about 42 bits. 6*7
- A code based on our unigram model would require about 77 bits. 11*7
- ASCII uses an average of 24 bits per word (168 bits total)!
- So better language models can also give us better data compression: as elaborated by the field of **information theory**.

Perplexity

- LM performance is often reported as **perplexity** rather than cross-entropy.
- Perplexity is simply $2^{\text{cross-entropy}}$
- The average branching factor at each decision point, if our distribution were uniform.
- So, 6 bits cross-entropy means our model perplexity is $2^6 = 64$: equivalent uncertainty to a uniform distribution over 64 outcomes.

Perplexity looks different in J&M 3^{rd} edition because they don't introduce cross-entropy, but ignore the difference in exams; I'll accept both!

Interpreting these measures

I measure the cross-entropy of my LM on some corpus as 5.2.
Is that good?

Interpreting these measures

I measure the cross-entropy of my LM on some corpus as 5.2.
Is that good?

- No way to tell! Cross-entropy depends on both the model and the corpus.
  - Some language is simply more predictable (e.g. casual speech vs academic writing).
  - So lower cross-entropy could mean the corpus is "easy", or the model is good.
- We can only compare different models on the same corpus.
- Should we measure on training data or held-out data? Why?

Sparse data, again

Suppose now we build a *trigram* model from Moby Dick and evaluate the same sentence.

- But I spent three never occurs, so $P_{MLE}(\text{three} \mid \text{I spent}) = 0$
- which means the cross-entropy is infinte.
- Basically right: our model says I spent three should never occur, so our model is infinitely wrong/surprised when it does!
- Even with a unigram model, we will run into words we never saw before. So even with short N-grams, we need better ways to estimate probabilities from sparse data.

Smoothing

- The flaw of MLE: it estimates probabilities that make the training data maximally probable, by making everything else (unseen data) minimally probable.
- **Smoothing** methods address the problem by stealing probability mass from seen events and reallocating it to unseen events.
- Lots of different methods, based on different kinds of assumptions. We will discuss just a few.

Add-One (Laplace) Smoothing

- Just pretend we saw everything one more time than we did.

$$P_{\text{ML}}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

$$\Rightarrow \qquad P_{+1}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1})} \qquad ?$$

Add-One (Laplace) Smoothing

- Just pretend we saw everything one more time than we did.

$$P_{\mathrm{ML}}(w_i|w_{i-2},w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)}{C(w_{i-2},w_{i-1})}$$

$$\Rightarrow \qquad P_{+1}(w_i|w_{i-2},w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)+1}{C(w_{i-2},w_{i-1})} \qquad ?$$

- NO! Sum over possible w_i (in vocabulary V) must equal 1:

$$\sum_{w_i \in V} P(w_i|w_{i-2},w_{i-1}) = 1$$

- If increasing the numerator, must change denominator too.

Add-one Smoothing: normalization

- We want: $$\sum_{w_i \in V} \frac{C(w_{i-2},w_{i-1},w_i)+1}{C(w_{i-2},w_{i-1})+x} = 1$$
- Solve for x:

$$\begin{aligned}
\sum_{w_i \in V} (C(w_{i-2},w_{i-1},w_i)+1) &= C(w_{i-2},w_{i-1})+x \\
\sum_{w_i \in V} C(w_{i-2},w_{i-1},w_i) + \sum_{w_i \in V} 1 &= C(w_{i-2},w_{i-1})+x \\
C(w_{i-2},w_{i-1})+v &= C(w_{i-2},w_{i-1})+x \\
v &= x
\end{aligned}$$

where v = vocabulary size.

Add-one example (1)

- *Moby Dick* has one trigram that begins with I spent (it's I spent in) and the vocabulary size is 17231.
- Comparison of MLE vs Add-one probability estimates:

	MLE	+1 Estimate
$\hat{P}$(three | I spent)	0	0.00006
$\hat{P}$(in | I spent)	1	0.0001

- $\hat{P}(\text{in}|\text{I spent})$ seems very low, especially since in is a very common word. But can we find better evidence that this method is flawed?

Add-one example (2)

- Suppose we have a more common bigram w_1, w_2 that occurs 100 times, 10 of which are followed by w_3.

	MLE	+1 Estimate
$\hat{P}(w_3|w_1,w_2)$	$\frac{10}{100}$	$\frac{11}{17331}$
		≈ 0.0006

- Shows that the very large vocabulary size makes add-one smoothing steal *way* too much from seen events.
- In fact, MLE is pretty good for frequent events, so we shouldn't want to change these much.

Add-α (Lidstone) Smoothing

- We can improve things by adding $\alpha < 1$.

$$P_{+\alpha}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + \alpha}{C(w_{i-1}) + \alpha v}$$

- Like Laplace, assumes we know the vocabulary size in advance.
- But if we don't, can just add a single "unknown" (UNK) item, and use this for all unknown words during testing.
- Then: how to choose α?

Optimizing α (and other model choices)

- Use a three-way data split: **training** set (80-90%), **held-out** (or **development**) set (5-10%), and **test** set (5-10%)
  - Train model (estimate probabilities) on training set with different values of α
  - Choose the α that minimizes cross-entropy on development set
  - Report final results on test set.
- More generally, use dev set for evaluating different models, debugging, and optimizing choices. Test set simulates deployment, use it only once!
- Avoids overfitting to the training set and even to the test set.

Better smoothing: Good-Turing

- Previous methods changed the denominator, which can have big effects even on frequent events.
- Good-Turing changes the numerator. Think of it like this:
  - MLE divides count c of N-gram by count n of history:

$$P_{\mathrm{ML}} = \frac{c}{n}$$

  - Good-Turing uses **adjusted counts** c^* instead:

$$P_{\mathrm{GT}} = \frac{c^*}{n}$$

Good-Turing in Detail

- Push every probability total down to the count class below.
- Each *count* is reduced slightly (Zipf): we're discounting!

c	N_c	P_c	P_c[*total*]	$c*$	$P*_c$	$P*_c$ [*total*]
0	N_0	0	0	$\frac{N_1}{N_0}$	$\frac{\frac{N_1}{N_0}}{N}$	$\frac{N_1}{N}$
1	N_1	$\frac{1}{N}$	$\frac{N_1}{N}$	$2\frac{N_2}{N_1}$	$\frac{2\frac{N_2}{N_1}}{N}$	$\frac{2N_2}{N}$
2	N_2	$\frac{2}{N}$	$\frac{2N_2}{N}$	$3\frac{N_3}{N_2}$	$\frac{3\frac{N_3}{N_2}}{N}$	$\frac{3N_3}{N}$

- c: count
  N_c: number of different items with count c
  P_c: MLE estimate of prob. of that item
  P_c[*total*]: MLE total probability mass for *all* items with that count.
  $c*$: Good-Turing smoothed version of the count
  $P*_c$ and $P*_c$ [*total*]: Good-Turing versions of P_c and P_c[*total*]

Some Observations

- Basic idea is to arrange the discounts so that the amount we *add* to the total probability in row 0 is matched by all the discounting in the other rows.
- Note that we only know N_0 if we actually know what's missing.
- And we can't always estimate what words are missing from a corpus.
- But for bigrams, we often assume $N_0 = V^2 - N$, where V is the different (observed) words in the corpus.

Good-Turing Smoothing: The Formulae

Good-Turing discount depends on (real) adjacent count:

$$\begin{aligned} c* &= (c+1)\tfrac{N_{c+1}}{N_c} \\ P*_c &= \tfrac{c*}{N} \\ &= \tfrac{(c+1)\frac{N_{c+1}}{N_c}}{N} \end{aligned}$$

- Since counts tend to go down as c goes up, the multiplier is < 1.
- The sum of all discounts is $\frac{N_1}{N_0}$. We need it to be, given that this is our GT count for row 0!

Good-Turing for 2-Grams in Europarl

Count	Count of counts	Adjusted count	Test count
c	N_c	c^*	t_c
0	7,514,941,065	0.00015	0.00016
1	1,132,844	0.46539	0.46235
2	263,611	1.40679	1.39946
3	123,615	2.38767	2.34307
4	73,788	3.33753	3.35202
5	49,254	4.36967	4.35234
6	35,869	5.32928	5.33762
8	21,693	7.43798	7.15074
10	14,880	9.31304	9.11927
20	4,546	19.54487	18.95948

t_c are average counts of bigrams in test set that occurred c times in corpus: fairly close to estimate c^*.

Good-Turing justification: 0-count items

- Estimate the probability that the next observation is previously unseen (i.e., will have count 1 once we see it)

$$P(\text{unseen}) = \frac{N_1}{n}$$

This part uses MLE!

- Divide that probability equally amongst all unseen events

$$P_{\mathrm{GT}} = \frac{1}{N_0}\frac{N_1}{n} \quad \Rightarrow \quad c^* = \frac{N_1}{N_0}$$

Good-Turing justification: 1-count items

- Estimate the probability that the next observation was seen once before (i.e., will have count 2 once we see it)

$$P(\text{once before}) = \frac{2N_2}{n}$$

- Divide that probability equally amongst all 1-count events

$$P_{\mathrm{GT}} = \frac{1}{N_1}\frac{2N_2}{n} \qquad \Rightarrow \qquad c^* = \frac{2N_2}{N_1}$$

- Same thing for higher count items

Summary

- We can measure the relative goodness of LMs on the same corpus using cross-entropy: how well does the model predict the next word?

- We need smoothing to deal with unseen N-grams.

- Add-1 and Add-α are simple, but not very good.

- Good-Turing is more sophisticated, yields better models, but we'll see even better methods next time.